Improving the design cycle for nanophotonic components

M. Fiers[1], E. Lambert, S. Pathak, B. Maes, P. Bienstman, W. Bogaerts

Photonics Research Group (INTEC), Ghent University - IMEC, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium

Abstract

We propose a software framework that greatly simplifies the design of nanophotonic components. While we illustrate the application in photonics, its benefits are applicable to other domains. In this approach, all steps in the workflow are based on a single high-level definition of the component, in a Python script. This is less error prone because there is only one high-level description, and the reproducibility is greatly improved. Furthermore it enables easy closed-loop modeling of components and circuits. Also, previous work can easily be built upon because lower level blocks can easily be replaced by new blocks. Such a framework can be used as a platform spanning multiple research groups to evaluate results.

Keywords: Nanophotonics, Designing components, Modelling components, Parametrized cell, Python

1. Introduction

In a typical research or design environment, fabrication of micro-and nanoscale devices is an expensive process with long turnaround times. Prior to submitting a design for fabrication, these devices are typically modelled and simulated in software. For example, in the field of nanophotonics, electromagnetic simulations are used to calculate how light propagates through such a device. Additionally virtual fabrication is used to compare how the designed structure deviates from the physical structure by modeling the fabrication errors and limitations. One major difficulty that arises when designing these devices is that each tool has its own user interface and moreover has its own representation to define components. Defining these devices in different tools is a tedious job, and

Email addresses: martin.fiers@intec.ugent.be (M. Fiers), emmanuel.lambert@intec.ugent.be (E. Lambert), shibnath.pathak@intec.ugent.be (S. Pathak), bjorn.maes@umons.ac.be (B. Maes), peter.bienstman@ugent.be (P. Bienstman), wim.bogaerts@ugent.be (W. Bogaerts)

[1]Corresponding author.

the risk to introduce errors in the specification of the device in each tool becomes rather high. The main characteristic of our approach is that one defines a component only once on a high level, and then extracts the necessary representations (e.g. a discretized matrix representing the component, a cross-section, a list of polygons ...) to drive the different simulation tools. Because only one definition exists, the transition to different simulation tools can be written once in a generic way which makes simulations much less error prone. It is also much easier to reproduce earlier results and to change sub-parts of the design. In this way, many variations can easily be compared to one another (e.g. different simulation methods, an improved component...).

Python is our programming language of choice. The main reason for using this programming language is the flexibility which it offers: it can be used to make very complex software implementations, yet it has a low threshold for researchers without programming skills.

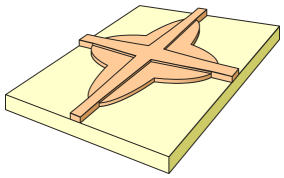
The basic class of the IPKISS software framework[1] is a parameterized cell (PCell), which is a concept that originates from the design of electronic circuits. All components inherit from this basic class. Different classes are available for visualization, simulation, exporting to different file formats... For this we use so-called mixins. An additional base class can dynamically be mixed into the PCell, which instantly makes all PCells inherit from these new base classes.

The paper is structured as follows: as the reader might not be familiar with photonics, we very briefly describe this specific research field in section 2. In section 3, we illustrate a typical workflow, i.e. the steps needed to design a component. We show which design problems typically arise and demonstrate how the software framework improves this flow. In section 4, the technical design and implementation of the framework is described, and in the last section we discuss an example which is typical for the nanophotonic research domain. However, it is easy to extend this architecture beyond the horizon of photonics: electronic design, multiphysics...

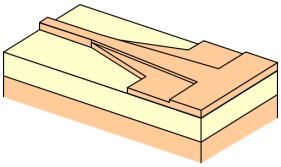
2. Photonics

Photonics is the field of controlling and manipulating light (photons) by means of optical components. This is in contrast to electronics, in which electrons are the information carriers. Some examples of photonic devices are: lasers, optical receivers and transmitters, CD/DVD drives, LED lighting... A recent trend in photonics is the drive towards miniaturization of components, and integrating many of them on a single chip. These so-called nano-photonic devices have a better performance, are more robust, and consume less power. One excellent material for making optical chips is silicon. Silicon has very low losses in the wavelength range that is used (near-infrared: 1300 nm and 1550 nm are commonly used). Fortunately, silicon is already used a lot in electronic chip fabrication, so we can reuse standard CMOS technology to manufacture photonic chips. In this technology, the silicon on insulator (SOI) wafer is patterned using deep UV lithography [2]. This opens the door to wafer-scale fabrication

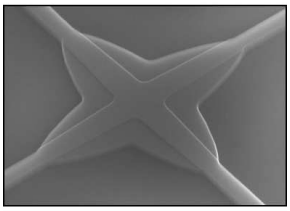
of nanophotonic chips, leading to very cheap devices. A few subcomponents of a nanophotonic circuit are displayed in figure 1.



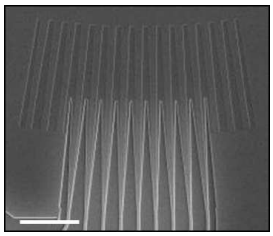
(a) Crossing



(b) Taper



(c) SEM image of a crossing



(d) SEM image of a taper

Figure 1: Some examples of some nanophotonic subcomponents.

3. Workflow for designing an optical component

3.1. Classical workflow

To illustrate the problems associated with our old workflow (that is, before adopting the framework) we show how in the past a device for splitting light would be modelled. The device itself is called a multimode interferometer (MMI) and is illustrated in figure 2.

The light that enters the input waveguide has a certain *mode profile*, i.e. the spatial distribution of the light coming in. For interpreting the propagation of light inside this waveguide, it is useful to know this profile. If the light source that enters this waveguide does not match this profile, a lot of light will scatter out of the waveguide and be lost.

To calculate the mode profile, an eigenmode Maxwell solver is used, called CAMFR [3]. The mode has a gaussian-like profile, as shown in figure 3 on the left. Because this is a 1D problem, it is rather straightforward to simulate this, it involves only a few parameters: the refractive index of the waveguide and its surroundings, and the width of the waveguide.

This mode is then used as an input source for a full time domain simulation in two dimensions. The tool used here is called Meep and is a finite difference time domain simulator [4]. With this tool, the fields can be calculated at all positions and at all times. Although Meep is a very powerful simulation tool, it is quite complex to calculate the mode profile with it. Yet we want to use this previously calculated mode profile because then we do not have to deal with the scattering problems coming from coupling light in the input waveguide.

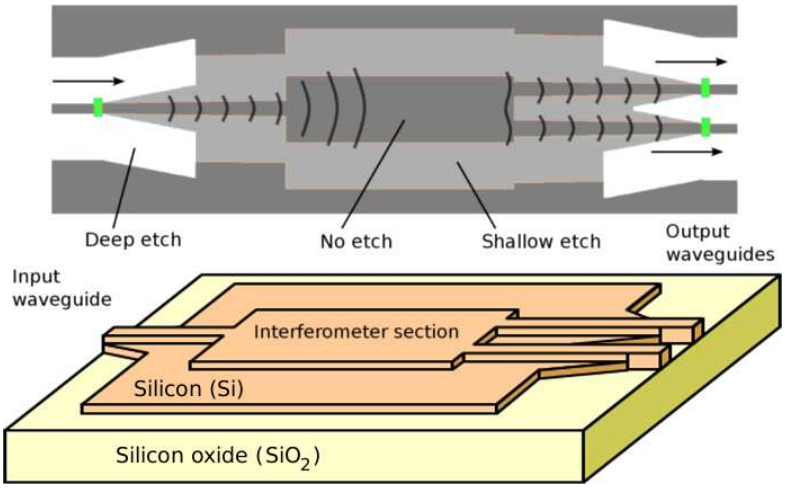


Figure 2: Schematic of a Multimode Interferometer. Light is split by the device. Different processing layers are shown, each with its own etch depth (no etch, shallow etch...). The input and output ports are shown in green.

This simplifies the interpretation of the FDTD simulation and speeds up the simulation, because a smaller simulation domain can be used.

Unfortunately, it is rather tedious to exchange information between these tools: the output of different tools usually have different file formats and/or the way this data is loaded into the simulation is different. Users will prefer to avoid these problems and use a more simplified approach to model their devices. Defining the component in Meep is done using several low level parameters, which have to be set manually. Making a geometry in Meep is usually done using the Scheme language. This involves some programming, and experience learns that it takes some iterations before the device is represented correctly.

In figure 3 on the right, one can see that the light is split nicely over the two outputs. After this simulation, the transmission can be calculated (ideally, it would be 50% for each port, but in practice there will always be some losses). This is usually done by loading the results from the output and importing them in a spreadsheet tool for visualization, or importing them in Matlab for data processing (curve fitting...).

PCell	Cross section for mode profile calculation	Complete geometry for FDTD simulation
2D simulation	CAMFR	Meep FDTD
3D simulation	FimmWave	Meep FDTD

Table 1: Simulation tools used for a 2D and 3D simulation.

After this step, a 3D simulation can be done to get more accurate results. This means the mode profile has to be calculated again with another tool that

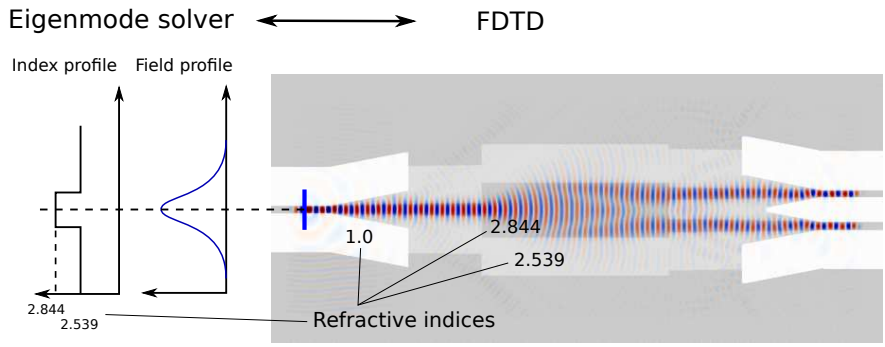


Figure 3: Two commonly used simulation tools: finite difference time domain (FDTD), and eigenmode solver.

is more appropriate for calculating modes for 2D cross sections. We use a commercial software tool for this, called Fimmwave [5]. The different simulation tools are listed in table 1. However, making a 2D cross-section of the waveguide is necessary and requires learning a new user interface. This mode profile must then be loaded in the tool for 3D FDTD simulation: again we use Meep. This means a new script has to be made to define the 3D geometry, and it requires checking whether the two scripts represent exactly the same component. This also means that the profile has to be exported from Fimmwave to Meep, which in practice is an inconvenient process.

In this design flow, we did not yet take into account processing variations and limitations. Especially, sharp bends and very tiny structures (on the order of 10-100 nm) will be different from the original design after fabrication. Including a virtual fabrication can improve the accuracy of the simulation results, but as long as the design process is not automated, it would be very cumbersome to include virtual fabrication.

3.2. New workflow

The new workflow that we developed in the past years and recently adopted, is much more automated by having only a single representation of the component in a high-level Python script. The workflow is depicted in figure 4 and incorporates the following steps:

Definition of the component. The basic description of a component is based on the mask layout required for fabrication in semiconductor technology. From these layouts, a library of components is made, which is called Picazzo. This library contains a lot of high-level components, such as the splitter from figure 2, waveguides to guide the light, grating couplers to guide light in and out of the optical chip... Here's an example:

Listing 3.2.1: Creation of the component

```
mmi = MmiSimple(length=7.780, width=2.920)
```

All these components are PCells, and all the following steps in the workflow will mix in new classes to enhance the functionality of this PCell.

Virtual fabrication and visualization. After defining the component, a visualization of the virtually fabricated device is made. In this stage one can already check for design errors, and possibly account for approximations made because of physical limitations. This can improve the accuracy between simulations and physical measurements and was not done in the old workflow.

Simulation. A next step involves simulation of the physical behaviour of the device. Interfacing to the various simulation tools is done automatically, so it is unnecessary to learn additional user interfaces. There is also no need to manually specify the geometry of the component, instead the high-level description of the Picazzo component is used to derive this geometry. Also, the interfacing between the different tools is done automatically. For example, the mode profile is calculated in CAMFR and then passed on to Meep, without any additional programming work.

Because simulation is the most CPU intensive part of the workflow, we made it possible to send simulations easily to a cluster. This is done by persisting the high-level design to a file. This file can be sent to a simulation cluster, and using the same framework, it can be modelled there, without the user needing to worry about the specific details on how to run a simulation on a large cluster. In the next code example we will create a simulation object. As explained before, a gaussian mode profile is used as input (`sources`). Some detectors are then added (`datacollectors`).

Listing 3.2.2: Creating the sources, and defining the simulation.

```
sources = [ModeProfileAtPort(center_wavelength=1550,
                        port=source_port,amplitude=0.1)]
datacollectors = [Fluxplane(port=input_port,
                        name="Flux at input port"),
                  Fluxplane(port=output_port,
                        name="Flux at output port"]
sim = mmi.create_simulation(
                  engine=MeepSimulationEngine(resolution=36,
                        sources,datacollectors))
sim.run()
```

During the simulation, data is stored in numerous formats. Using Python, this data can be visualized easily, for example the electromagnetic field in a cross-section of the nanophotonic component. In section 4 we show which tools are used for this.

Data processing. We now have the output data available in Python, and can use this for all kinds of post-processing, curve fitting, parameter extraction... This post-processing used to be done in various external environments (Excel, Matlab ...), which involved a lot of manual data conversion for each tool.

Re-iterate. Depending on the results of the data processing, parameters can be changed easily, and the user can repeat the cycle until the desired behaviour for the component is reached. In the old workflow, much more manual actions were required before the same set of operations in the workflow could be repeated.

Physical fabrication. Eventually, this design is exported to the GDS format, which is a format used to send designs to a mask shop. From this GDS file, a physical component is made. This is illustrated in figure 5.

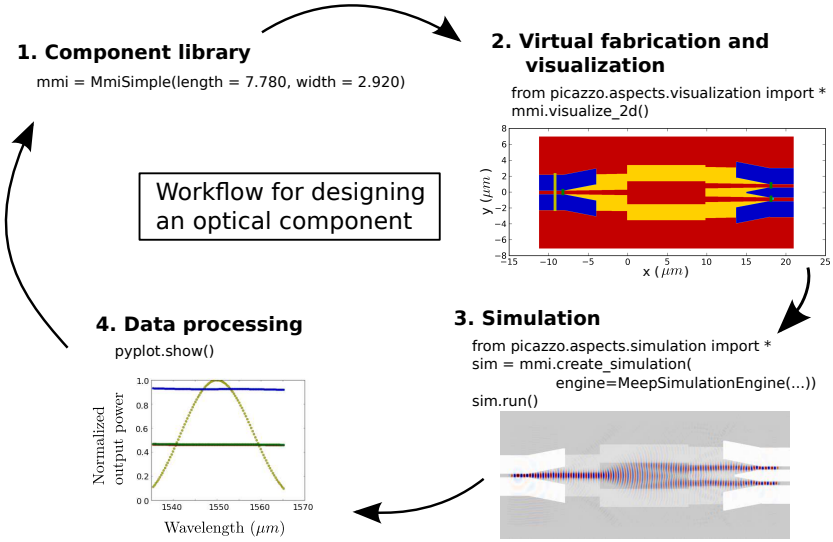


Figure 4: New workflow: designing a multimode interferometer (MMI). All steps are gathered in a single high-level script.

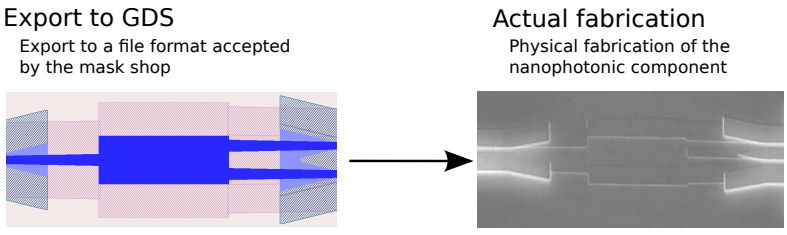


Figure 5: Example of fabrication steps for a multimode interferometer.

4. Design and implementation of the framework

This section explains how we designed and implemented IPKISS. The PCell, the core concept of the framework, is explained, and it is demonstrated how mixins are used to flexibly add functionality to these PCells.

The proposed software framework is based on the language Python. The programming language has to meet several requrements. First, the requirements for the software developer, who demands a language that can be used to make complex software, and second, the researcher, who does not want to bother about all technical details of the implementation, and wants a clean scripting environment. Furthermore, the ability to integrate different tools in the framework is important. For example, some C/C++ -code has to be run from within the framework. In Python, there are several ways to interface to C/C++, for example using SWIG [6].

4.1. PCell

The core of the framework is a parametrized cell (PCell) engine. PCell is a concept widely used in the automated design of electronic circuits. Basically it is a class which is used to represent physical entities such as a transistor, a resistor... In our software framework, we call this basic entity a **Structure**, and it represents a nanophotonic component. A structure has some `Properties` that describe the object. The following piece of simplified code illustrates the PCell concept when making the MMI (see figure 2):

Listing 4.1.1: Making the MMI.

```
class MmiSimple(Structure):
    __name_prefix__ = "MmiSimple"
    width = PositiveNumberProperty(default=10.0, required = True)
    height = PositiveNumberProperty(default=5.0, required = True)
    area = PositiveNumberProperty(doc="Area under the MMI")
    def define_area(self):
       return self.width*self.height
    def define_layout(self,layout):
       # Construct the MMI using rectangles and triangles on different
       # processing layers (see figure 2) with different etch depths.
       layout += Rectangle(layer=ProcessLayer(TECH.PROCESS.NO_ETCH),
                           (0.0, 0.0),
                           (self.width, self.height))
       layout += Rectangle(layer=ProcessLayer(TECH.PROCESS.SHALLOW),
                           (0.0, 0.0),
                           (self.width + 2*self.trench_width,
                            self.height + 2*self.trench_width))
       ...
       return layout
```

In Python, variables do not have to be declared with a certain type. This has the drawback that type errors are not caught upon initialization, but much

later, when these variables are actually used. E.g. when multiplying two strings, an exception is thrown and a stack trace is displayed. This stack trace can be very intimidating for a novice user. For that reason, we choose to give the user immediate and clear feedback on the validity of the arguments upon initialization of a Pcell.

To achieve this, we use `PropertyDescriptor` to type each variable. For example, `PositiveNumberProperty` inherits from `PropertyDescriptor` and looks like this:

Listing 4.1.2: `PositiveNumberProperty` as a `PropertyDescriptor`

```
def PositiveNumberProperty(restriction=None, **kwargs):
    R = RESTRICT_NUMBER & RESTRICT_POSITIVE & restriction
    return PropertyDescriptor(restriction=R, **kwargs)
```

Upon initialization, the `PropertyDescriptor.__set__(self, obj, value)` function checks whether the value assigned matches the type required, and displays an error when the type expected does not match the wanted type. Combining restrictions is possible, for example RestrictType(list) & RestrictLength(0,5) requires the variable to be a list, with a length between 0 and 5. Additionally, the PropertyDescriptor can be set using the define function. For example, the variable `area` in the example above is associated automatically with the function `define_area`.

`Structure` inherits from a class `StrongPropertyInitializer` which checks whether the correct keyword arguments are passed during initialization. This mechanism makes sure no redundant or wrong information is passed during construction. It also removes the need for an `__init__` function if the only need is to assign arguments to class members. Properties use descriptors to change the default behaviour for getting and setting properties. In this way, values can be cached, and checked for correct types. Each property optionally has a `.doc` member which is used to generate automated documentation of components. This has been used successfully to generate documentation for all available components in our library.

In the Picazzo library, there are a lot of designs for already fabricated and tested devices. Using little programming work, new components can be designed. The flexibility of Python allows to easily swap and redesign pieces of components.

In parallel to the development of our PCell class, other powerful libraries were developed that allow typing of Python variables, support delegation and initialization of variables... One of these is the Traits library, developed at Enthought [7]. The functionality of this library is very similar to the functionality we provide, and the possibility exists that we will migrate to the Traits library in the future.

4.2. Mixins

The PCell engine is enriched with additional functionalities by mixing in additional classes into it. After mixing in a class, the PCell inherits from this

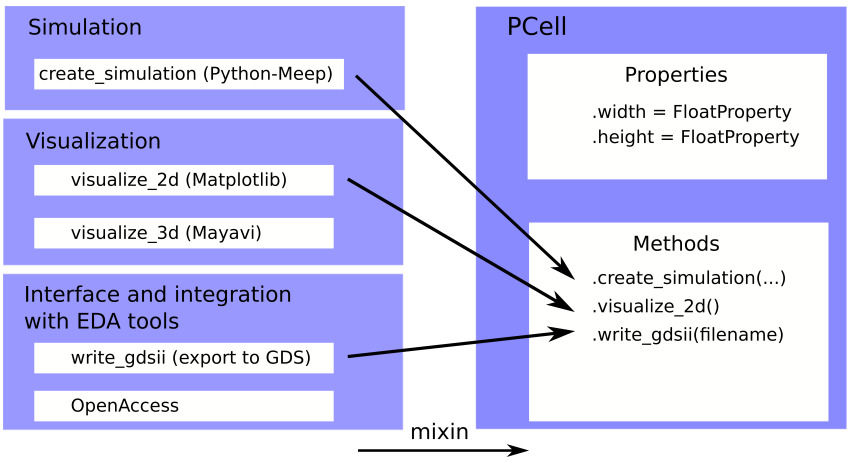


Figure 6: In the left, methods of classes are shown. These classes can dynamically be mixed into the PCell.

class. This is used to add functionality such as generating a representation of the physical layout, visualizing, simulations, and interfacing to external tools. This is illustrated in figure 6. There are several reasons to use mixins rather than to inherit all classes explicitly (multiple inheritance). First of all, it reduces the complexity of the PCell class. In this way you do not pollute the userspace with functionalities that will never be used. Second, new modules can simply be plugged in without changing the code base. Third, it is a way to protect intellectual property. Additional functionality can be part of a proprietary module and can easily be plugged into the framework if allowed.

Mixins are realized in Python by changing the special attribute `__bases__`, which is a member of the class object. It contains the list of classes which the parent class inherits from. This `__bases__` class can be modified dynamically, allowing to mix in other classes at runtime. When mixing in a class, all existing instances of the parent class automatically receive this functionality. Mixin base classes are typically mixed in only when the users script loads a specific module. This is implemented as follows:

Listing 4.2.1: Implementation of mixins

```
class MetaMixinBowl(type):
    def mixin(cls, mixin_class):
        if not mixin_class in cls.__bases__:
            if cls.__bases__ == (object,):
                cls.__bases__  =  (mixin_class,)
            else:
                cls.__bases__  =  (mixin_class,) + cls.__bases__

class MixinBowl(object):
    __metaclass__ = MetaMixinBowl
```

```
# This is the PCell object
class Structure(MixinBowl):
    ...
```

Here we introduce the use of metaclasses. A metaclass is a class whose instances are classes. They are often used to override object creation, but here it is only used to add functions to the Structure class. In this case, Structure is an instance of MetaMixinBowl, and as such, `Structure.mixin` is a valid function. At this point, mixing in a class Foo is as simple as `Structure.mixin(Foo)`.

4.3. Interfacing with different simulation tools

In order to interface to different simulation tools it was necessary to model several entities in the core of our software framework.

- Structure. This is the PCell object, the basic class on which our framework is based. It allows to check for variable types and supports the mixing in of other classes, such as the simulation class. Structure objects are stored in a library and have a unique identifier with which they can be retrieved. The library contains no duplicate structures, which is interesting when a certain structure is repeated a lot of times.

- The physical concepts. The link between the PCell object and the simulation tools is the geometric representation of the device, consisting of different materials. Each material has its own physical properties such as refractive index, a temperature coefficient, a doping profile, a stress and strain matrix... These physical concepts are the least common denominator of the software framework.

- Abstract models for the different simulation types. There are different type of solvers. For example in optics, a mode solver calculates the electromagnetic field distribution in a cross-section of a component, and a finite difference time domain (FDTD) solver calculates the electromagnetic field at all positions as a function of time, given an input light source. The representation of the field is different for a mode solver (sum of eigenmodes), than for a FDTD simulation (field at all times), and the abstract model takes care of the appropriate conversions. This was already illustrated in figure 3.

- From the abstract simulation models, concrete implementations are inherited for the specific tools used (for example in optics, we use Meep as FDTD solver, CAMFR and FimmWave as eigenmode solvers). Additional implementations can be developed for other research domains. This is an important investment in the framework which was a steep threshold in the beginning, before we could start with the technical integration of different tools. The advantage is that all scripts can now be written as function of these abstract classes, and the user can flexibly switch between tools.

It becomes clear that implementing an interface to an external tool demands a lot of investment and technical expertise. It also requires good knowledge of the simulation tools because an interface should be based on best practice. An experienced researcher which is familiar with the scripting tools or user interface can easily set up a simulation. When implementing an interface to a tool, this know-how becomes part of the framework. Not only will researchers save time to define components in different languages and make simulations less error prone, they also do not have to care about learning a wide variety of scripting tools or user interfaces. The know-how of fellow researchers can be easily leveraged in this way, so that researchers can focus on their core research activity.

Interfacing with different tools is not trivial: This depends on the specific implementation of the tool. CAMFR, for example, is scripted in Python, so integrating this in the toolbox does not add technical difficulties, apart from converting the nanophotonic component to the necessary syntax. For some tools, it is possible to communicate with them using sockets (for example: Fimmwave). Another way to interface to a tool is by using the tool's API. If an API is provided, one can tightly integrate the tool with the framework. A very good example of this is Python-meep [8]. SWIG was used to make the bridge between the C++ program Meep and Python. In this way, scripts could directly be written in python instead of using the C++ API or the Scheme language. When interfacing directly is not possible, one can still interface through files.

Another mixin has been developed that makes the PCell available to OpenAccess compliant tools, such as Cadence, which is an EDA tool commonly used to design electronic systems.

4.4. Python libraries

The rich ecosystem of Python greatly facilitates research activities. We list some of the employed libraries together with their use. We use Mayavi [9] for 3D visualization of the devices and Matplotlib [10] for 2D visualization. Shapely [11] is used to manipulate the geometry of the components with logical operations during the algorithm for virtual fabrication. h5py [12] is used to read data from simulations, and scipy is used for data fitting. Next to these free libraries, we also interface with commercial tools, e.g. FimmWave [5]. Our philosophy is to include at least a free tool where possible to cover the basic functionality without an additional cost. Other libraries can be added in the future. For consistency within our group, and to facilitate installation, we use the Enthought Python distribution [7], which contains many of these libraries, and is free for academic use.

4.5. Future work

One of the important additions that are planned are Netlists. This concept is indispensible in electronic design and in the near future, will be necessary in optical design as well. Suppose we have two subcomponents A and B, stored in the `children` of our structure, which should be linked together. Using the ports of A and B, we can link these together using a `Net`. The following pieces of code illustrates this:

Listing 4.5.1: Future work: netlists

```
class AB(Structure):
    ....
    def define_netlist(self, netlist):
        N = Net()    # A net can link an arbitrary amount of ports
        N += self.children.A.east_ports[0]
        N += self.children.B.west_ports[0]
        netlist += N
        return netlist
```

The structure AB now contains an internal representation of its network. This can be used to route electrical and/or optical signals from one Structure to the other.

5. Example: An Arrayed Waveguide Grating

In this paragraph, we show how we designed and modeled an optical component, which is our main research focus. We demonstrate how we can easily swap components, and how different simulation models can be used for different subcomponents. One can easily generalize the used concepts to other domains, including multiphysics simulations, electronic design...

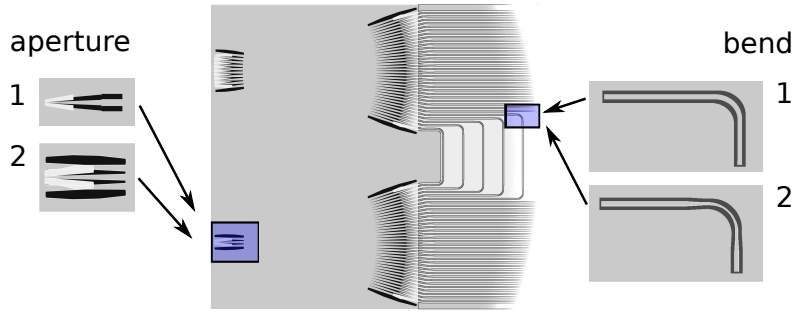


Figure 7: Schematic diagram of an AWG. Subcomponents can be replaced very easy, such as the input aperture (normal, MMI), and bends (normal, curved).

The component we study here is an Arrayed Waveguide Grating (AWG), see figure 7. It is one of the vital components in Wavelength Division Multiplexing (WDM) systems. They are used to separate many wavelength channels into different waveguides (or vice versa, merge them). It consists of two star couplers and an array of waveguides with a linear increment of length. A light beam enters the input star coupler and is distributed over the waveguide array. The different wavelengths reach the second star coupler with different phase shift. Because of this, different wavelengths focus at different output positions.

Using a single simulation technique it is difficult to simulate these kinds of complex structures. We developped a hybrid semi-analytical model using

the software framework to calculate the transmission matrices (T-matrix) for the different parts of the AWG. Multiplication of all these T-matrixes gives the T-matrix of the entire AWG. Different simulation strategies are used for the subcomponents. The aperture, for instance, is modelled in CAMFR, and the propagation through the waveguides is done analytically. From virtual fabrication of the aperture (Figure 8(a)) we obtain a geometry for the CAMFR simulation (Figure 8(b)).

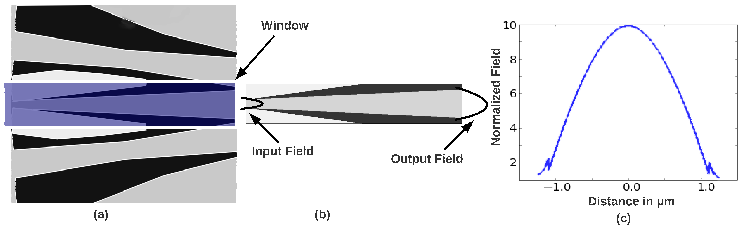


Figure 8: (a) Aperture of the star coupler. (b) Selected portion of the aperture used as a CAMFR stack. (c) Mode profile of the aperture.

This is then combined with a T-matrix model of the array waveguides and a simulation of the star coupler to render a complete simulation of the AWG. Figure 8(c) shows the simulation result of the aperture, and figure 9(a) shows the result of the complete simulation.

As figure 7 suggests, it is very easy to plug in other subcomponents in the AWG. Two possible replacements are

- Another aperture for the AWG. Using an aperture that looks like an MMI (see section 3), we can improve the bandwidth of the component [13]. This can be seen in figure 9(b).
- Improved bends. Recently, new bend strategies were developed that give rise to less loss. By including them in the script, all new fabricated devices will contain these improved bends.

Instead of writing separate independent scripts that do all this work, we were able to describe and solve the problem fully in the software framework. This base script can be used by other scientists to optimize and fabricate the component.

6. Conclusion

The IPKISS software framework provides a powerful and generic environment for designing, simulation and fabrication of electronical and optical circuits. We have demonstrated how the software framework improves the workflow for designing components. We have used the framework successfully to make complex networks of optical components and to model subparts of these networks in a structured and reproducible way. In the future, more simulation

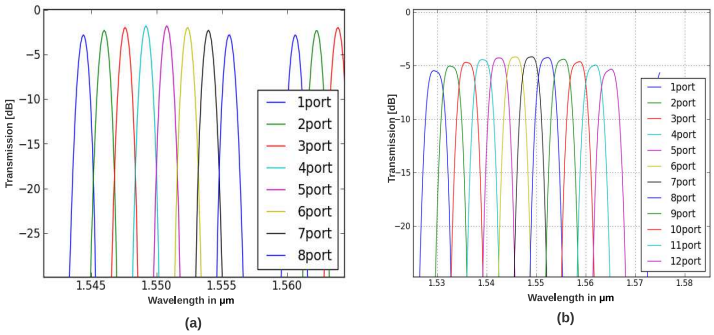


Figure 9: Simulation result of the AWG, using aperture 1 (a), using aperture 2 (b). The flat tops in (b) are a great improvement.

methods will be added to the framework, and netlists will be implemented to enable the linking of ports. The latter will enable linking of different simulation strategies, and circuit simulations become possible. Many optical chips have already been fabricated by using this software framework.

[1] `http://www.ipkiss.org` .

[2] S. Selvaraja, P. Jaenen, W. Bogaerts, P. Dumon, D. V. Thourhout, R. Baets, Fabrication of Photonic Wire and Crystal Circuits in Silicon-on-Insulator Using 193nm Optical Lithography, Journal of Lightwave Technology 27(18), pp 4076–4083.

[3] `http://www.photond.com/products/fimmwave.htm` .

[4] A. F. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J. D. Joannopoulos, S. G. Johnson, MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method, Computer Physics Communications 181 (2010) 687–702, doi:doi:10.1016/j.cpc.2009.11.008.

[5] `http://camfr.sourceforge.net` .

[6] `http://www.swig.org` .

[7] `http://www.enthought.com` .

[8] E. Lambert, M. Fiers, S. Nizamov, M. Tassaert, W. Bogaerts, Python bindings for the open source electromagnetic simulator MEEP, Computing in Science and Engineering (accepted for publication).

[9] `http://mayavi.sourceforge.net` .

[10] `http://matplotlib.sourceforge.net` .

[11] `http://trac.gispython.org/lab/wiki/Shapely` .

[12] `http://h5py.alfven.org` .

[13] S. Pathak, W. Bogaerts, E. Lambert, P. Dumon, D. V. Thourhout, Integrated Design and Simulation Tools for Silicon Photonic Arrayed Waveguide Gratings, Annual Symposium of the IEEE Photononics Benelux Chapter, p. 41–44, 2010.